SYSTEM, METHOD, AND COMPUTER PROGRAM FOR FLOW CONTROL IN A DISTRIBUTED BROADCAST-ROUTE NETWORK WITH RELIABLE TRANSPORT LINKS

Inventor:

Serguei Y. Osokine

Field of the Invention

This invention pertains generally to systems and methods for communicating information over an interconnected network of information appliances, and more particularly to system and method for controlling the flow of information over a distributed information network having broadcast-route network and reliable transport link network characteristics.

BACKGROUND

The Gnutella network does not have a central server and consists of the number of equal-rights hosts, each of which can act in both the client and the server capacity. These hosts are called 'servents'. Every servent is connected to at least one other servent, although the typical number of connections (links) should be more than two (the default number is four). The resulting network is highly redundant with many possible ways to go from one host to another. The connections (links) are the reliable TCP connections.

When the servent wishes to find something on the network, it issues a request with a globally unique 128-bit identifier (ID) on all its connections, asking the neighbors to send a response if they have a requested piece of data (file) relevant to the request. Regardless of whether the servent receiving the request has the file or not, it propagates (broadcasts) the request on all other links it has, and remembers that any responses to the request with this ID should be sent back on the link which the request has arrived from. After that if the request with the same

ID arrives on the other link, it is dropped and no action is taken by the receiving servent in order to avoid the 'request looping' which would cause an excessive network load.

Thus ideally the request is propagated throughout the whole Gnutella network (GNet), eventually reaching every servent then currently connected to the network. The forward propagation of the requests is called 'broadcasting', and the sending of the responses back is called 'routing'. Sometimes both broadcasting and routing are referred to as the 'routing' capacity of the servent, as opposed to its client (issuing the request and downloading the file) and server (answering the request and file-serving) functions. In a Gnutella network each node or workstation acts as a client and as a server.

Unfortunately the propagation of the request throughout the whole network might be difficult to achieve in practice. Every servent is also the client, so from time to time it issues its own requests. Thus if the propagation of the requests is unlimited, it is easy to see that as more and more servents join the GNet, at some point the total number of requests being routed through an average servent will overload the capacity of the servent physical link to the network.

Since the TCP link used by the Gnutella servents is reliable, this condition manifests itself by the connection refusal to accept more data, by the increased latency (data transfer delay) on the connection, or by both of these at once. At that point the Gnutella servent can do one of three things: (i) it can drop the connection, (ii) it can drop the data (request or response), or (iii) it can try to buffer the data in hope that it will be able to send it later.

The precise action to undertake is not specified, so the different implementations choose different ways to deal with that condition, but it does not matter – all three methods result in serious problems for the Gnet, namely one of A, B, or C, as follows: (A) Dropping the connection causes the links to go up and down all the time, so many requests and responses are simply lost, because by the time the servent has to route the response back, the connection to route it to is no longer available. (B) Dropping the data (request or response) can lead to a response being dropped, which overloads the network by unnecessarily broadcasting the requests over hundreds of servents only to drop the responses later. (C) Buffering the data increases the latency even more. And since it does little or nothing to fix the basic underlying problem (an attempt to transmit more data than the network is physically capable of) it only causes the servents to eventually run out of memory. To avoid that, they have to resort to other two ways of dealing with the connection overload albeit with much higher link latency.

These problems were at least somewhat anticipated by the creators of the Gnutella protocol, so the protocol has a built-in means to limit the request propagation through the network, called 'hop count' and 'TTL' (time to live). Every request starts its lifecycle with a hop count of zero and TTL of some finite value (de facto default is 7). As the servent broadcasts the request, it increases its hop count by one. When the request hop count reaches the TTL value, the request is not broadcast anymore. So the number of hosts **N** that see the request can be approximately defined by the equation:

$$(1) \qquad N = (\text{avLinks} - 1) \wedge \text{TTL}, \qquad \text{(EQ. 1)}$$

where avLinks is the average number of the servent connections, and the TTL is the TTL value of the request. For the avLinks = 5 and TTL = 7 this comes to a value of N of about 10,000 servents.

Unfortunately the TTL value and the number of links are typically hard-coded into the servent software and/or set by the user. In any case, there's no way for the servent to quickly (or dynamically) react to the changes in the GNet data flow intensity or the data link capacity. This leads to the state of affairs when the GNet is capable of functioning normally only when the number of servents in the network is relatively small or they are not actively looking for data. When either of these conditions is not fulfilled, the typical servent connections are overloaded with the negative consequences outlined elsewhere in this description. Put simply, the GNet enters the 'meltdown' state with the number of 'visible' (searchable from the average servent) hosts dropping from the range of between about 1,000-4,000 to a much smaller range or between about100-400 or less, which decreases the amount of searchable data by a factor of ten or about an order of magnitude. At the same time the search delay (the time needed for the request to traverse 7 hops (the default) or so and to return back as a response) climbs to hundreds of seconds. Response time on the order of hundreds of seconds are typically not tolerated by users, or at the very least are found to be highly irritating and objectionable.

In fact, the delay becomes so high that the servent routing tables (the data structures used to determine which connection the response should be routed to) reach the full capacity, overflow and time out even before the response arrives so that no response is ever received by the requestor. This, in turn, narrows the search scope even more, effectively making the Gnutella unusable from the user standpoint, because it cannot fulfill its stated goal of being the file searching tool.

The 'meltdown' described above has been observed on the Gnutella network, but in fact the basic underlying problem is deeper and manifests itself even with a relatively small number of hosts, when the GNet is not yet in an actual meltdown state.

The problem is that the GNet uses the reliable TCP protocol or connection as a transport mechanism to exchange messages (requests and responses) between the servents. Being the reliable vehicle, the TCP protocol tries to reliably deliver the data without paying much attention to the delivery latency (link delay). Its main concern is the reliability, so as soon as the data stream exceeds the physical link capacity, the TCP tries to buffer the data itself in a fashion, which is not controlled by the developer or the user. Essentially, the TCP code hopes that this data burst is just a temporary condition and that it will be able to send the buffered data later.

When the GNet is not in a meltdown state, this might even be true – the burst might be a short one. But regardless of the nature of the burst, this buffering increases the delay. For example, when a servent has a 40 kbits/sec modem physical link shared between four connections, every connection is roughly capable of transmitting and receiving about 1 kilobyte of data per second. When the servent tries to transmit more, the TCP won't tell the servent application that it has a problem until it runs out of TCP buffers, which are typically of about 8 kilobyte size.

So even before the servent realizes that its TCP connections are overloaded and has any chance to remedy the situation, the link delay reaches 8 seconds. Even if just two servents along the 7-hop request/response path are in this state, the search delay exceeds 30 seconds (two 8-second delays in the request path and two – in the response path). Given the fact that the GNet typically consists of the servents with very different communication capabilities, the probability is high that at least some of the servents in the request path will be overloaded. Actually this is exactly what can be observed on the Gnutella network even when it is not in the meltdown state despite the fact that most of the servents are perfectly capable of routing data with a sub-second delay and the total search time should not exceed 10 seconds.

Basically, the 'meltdown' is just a manifestation of this basic problem as more and more servents become overloaded and eventually the number of the overloaded servents reaches the 'critical mass', effectively making the GNet unusable from a practical standpoint.

It is important to realize that there's nothing a servent can do to fight this delay – it does not even know that the delay exists as long as the TCP internal buffers are not yet filled to capacity.

Some developers have suggested that UDP be used as the transport protocol to deal with this situation, however, the proposed attempts to use UDP as a transport protocol instead of TCP are likely to fail. The reason for this likely failure is that typically the link-level protocol has its own buffers. For example, in case of the modem link it might be a PPP buffer in the modem software. This buffer can hold as much as 4 seconds of data, and though it is less than the TCP one (it is shared between all connections sharing the physical link), it still can result in a 56-second delay over seven request and seven response hops. And this number is still much higher than the technically possible value of less than ten seconds and, what is more important, higher than the perceived delay of the competing Web search engines (such as for example AltaVista, Google, and the like), so it exceeds the user expectations set by the 'normal' search methods.

Therefore, there remains a need for a system, method, and computer program and communication protocol that minimizes the latency and reduces or prevents GNet or other distributed network overload as the number of servents grows.

SUMMARY

The invention provides improved data or other information flow control over a distributed computing or information storage/retrieval network. The flow, movement, or migration of information is controlled to minimize the data transfer latency and to prevent overloads. A first or outgoing flow control block and procedure controls the outgoing flow of data (both requests and responses) on the network connection and makes sure that no data is sent before the previous portions of data are received by a network peer in order to minimize the connection latency. A second or Q-algorithm block and procedure controls the stream of the requests arriving on the connection and decides which of them should be broadcast to the neighbors. Its goal is to make sure that the responses to these requests would not overload the outgoing bandwidth of this connection. A third or fairness block makes sure that the connection is not monopolized by any of the logical request/response streams from the other connections. It allows to multiplex the logical streams on the connection, making sure that every stream has its own fair share of the connection bandwidth regardless of how much data are the other streams capable of sending. These blocks and the functionality they provide may be used separately or in conjunction with each other. As the inventive method, procedures, and algorithms may advantageously be implemented as computer programs, such as computer programs in the form of software, firmware, or the like, the invention also advantageously provides a computer program and computer program product when stored on tangible media. Such computer programs may be executed on appropriate computer or information appliances as are known in the art, and may typically include a processor and memory couple to the processor.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a diagrammatic illustration showing an embodiment of a distributed information network providing flow control according to the invention.

DETAILED DESCRIPTION OF EMBODIMENTS OF THE INVENTION

The inventive system, method, and computer program solve the aforedescribed problems and limitations by minimizing the latency and preventing Gnutella and non-Gnutella distributed network overload as the number of nodes (clients, servers, or servents) grows. Almost all features of the inventive method and algorithm, except possible for some Gnutella specific version backward compatibility features, are not Gnutella-specific and can be utilized by any distributed network of similar architecture or topology. An exemplary embodiment of the invention is illustrated in FIG. 1 which shows an embodiment of a distributed information network on the form of a Gnutella network providing flow control according to the invention.

Aspects of the inventive system and method are directed to achieving large and in practical terms nearly infinite scalability of the distributed networks, which use a broadcast-route or analogous method to propagate the requests (such as information, data, or file requests) through the network. The broadcast-route as used here means a method of request propagation when the host broadcasts the request it receives on every connection it has except the one it came from and later routes the responses back to that connection.

The inventive method and algorithm is designed for the networks with reliable transport links. This primarily means the networks, which use TCP and TCP-based protocols (i.e. HTTP) for the data exchange, but the usefulness of the algorithm is not limited to TCP-based networks, and those workers having ordinary skill in the art will appreciate the applicability of the inventive system, method, and algorithm to other communication and signaling protocols that are expected to be developed and adopted in the foreseeable future. The method and algorithm might be used even for the 'unreliable' transport protocols like UDP, because they might (and do) utilize the reliable protocols on some level – for example, unreliable UDP packets can be sent over the modem link with the help of the reliable (PPP or its analog) protocol. Finally, as already indicated, the primary target of the algorithm is the Gnutella network (GNet), which is widely used as the distributed file search and exchange system.

The inventive system and method achieve large and in practical terms nearly infinite scalability of the distributed computing and information exchange networks, which use the 'broadcast-route' method to propagate the requests through the network. The 'broadcast-route' here means the method of the request propagation when the host broadcasts the request it receives on every connection it has except the one it came from and later routes the responses back to that connection. The inventive algorithm is designed for the networks with reliable transport links. This primarily means the networks, which use TCP and TCP-based protocols (i.e. HTTP) for the data exchange, but the usefulness of the algorithm is not limited to TCP-based networks. The algorithm might be used even for the 'unreliable' transport protocols like UDP, because they might (and do) utilize the reliable protocols on some level – for example, unreliable UDP packets can be sent over the modem link with the help of the reliable (PPP or its analog) protocol.

In spite of the generality of its applicability, the primary target of the algorithm is the Gnutella network, which is widely used as the distributed file search and exchange system. Its protocol specifications can be found as of 28 November 2000 on the Internet at:
http://gnutella.wego.com/go/wego.pages.page?groupId=116705&view=page&pageId=119598&folderId=116767&panelId=-1&action=view;
http://www.gnutelladev.com/docs/capnbry-protocol.html;

http://www.gnutelladev.com/docs/our-protocol.html; and http://www.gnutelladev.com/docs/gene-protocol.html .

The system, method, and algorithm dealing with the problems described above should satisfy several conditions: (i) It should make the Get highly scalable and in a practical sense as infinitely scalable as possible – that is, the large or practically infinite growth of the number of hosts in the network should not result in the network meltdown. (ii) It should bring the search delay to a technically reasonable minimum – that is, it should not result in an excessive request/response buffering when it is avoidable. (iii) It must be able to be used to connect the servents with various physical link capacities (even including asymmetrical links with different upload and download bandwidths where feasible) without causing the overload of the low-capacity links by the high-capacity ones. (iv) It should make a best-effort attempt to fairly share the connection between the logical request/response streams from other connections with different bandwidths, preventing the 'starvation' of the low-bandwidth streams because of the connection being monopolized by the high-rate logical streams. That also includes the ability of the algorithm to survive the denial-of-service (DoS) attacks, which are essentially the attempts of a single host or a group of hosts to monopolize the network, denying service for the legitimate users. (v) It should be backward-compatible with the already deployed Gnutella servent code base – even though the 'old' servents might be unable to fully reap the benefits of the algorithm, at the very least, they should be able to work with the 'new' servents without any quality of service degradation.

Actually almost all of the inventive method and procedure's features (except for the backward compatibility) are not Gnutella-specific and can be utilized by any distributed network with a similar 'broadcast-route' architecture as long as the network transport mechanisms include the reliability feature at least on one level.

One algorithmic or methodological goal of the invention is to control the flow of data through the Gnutella servent in order to minimize the data transfer latency and to prevent overloads, so it is called the 'flow control' algorithm. It consists of three basic blocks. A first or "outgoing flow control block" controls the outgoing flow of data (both requests and responses) on the connection and makes sure that no data is sent before the previous portions of data are received by the peer servent in order to minimize the connection latency. A second or "Q-algorithm block" controls the stream of the requests arriving on the connection and decides which of them should be broadcast to the neighbors. Its goal is to make sure that the responses to these requests would not overload the outgoing bandwidth of this connection. A third or "fairness block" makes sure that the connection is not monopolized by any of the logical request/response streams from the other connections. It allows to multiplex the logical streams on the connection, making sure that every stream has its own fair share of the connection bandwidth regardless of how much data are the other streams capable of sending. These blocks and the functionality they provide may be used separately or in conjunction with each other. For example, the flow control block may be used without the Q-algorithm block or the fairness block, and both the flow control block and the q-algorithm block may be used together without the fairness block. Each of the structural and algorithm features of these blocks are described in greater detail below.

Outgoing Flow Control Block

The Outgoing Flow Control Block controls the flow of data sent to a connection which has reliable transport at least somewhere in its network stack and tries to minimize the latency (transport delay) of this connection. It recognizes that some components of the transport delay cannot be controlled (physical limit determined by the speed of light, transport delay on the Internet routers, and the like). Still, the delays related to the buffering undertaken by the reliable transport layers (TCP, PPP, or the like) can be several times higher (seconds as opposed to hundreds of milliseconds), so their minimization can have dramatic effects on the transport delay on the connection.

The flow control block tries to use the simplest possible way of controlling the delay in order to decouple itself as much as possible from the specific buffering and retransmission mechanisms in the reliable transport, even at the cost of some theoretical connection throughput loss.

Thus several goals are achieved by the outgoing flow control block as now described. First, the algorithm can be used over a variety of transport mechanisms – not only TCP, but also HTTP over TCP, UDP over PPP, etc. This makes it possible to use the same algorithm if for some reason (for example, firewalls or something else) it would be necessary to migrate the distributed network 'broadcast-route' algorithm to a different transport protocol. Second, it makes the flow control algorithm not sensitive to a specific TCP implementation. Even though it is possible to optimize the flow control algorithm for an 'ideal' TCP implementation, it might be difficult to verify such an algorithm for every existing implementation of TCP, requiring the complicated and unreliable compatibility testing. Thus independence from a specific TCP implementation is valuable. Third, the 'decoupling' of the flow control algorithm from the underlying transport layer minimizes the possible feedback effects between it and the transport, making the algorithm more stable from a control theory standpoint.

In fact, very few assumptions are made about the nature of the underlying transport layers – even the reliability is not required. The algorithm mitigates possible negative effects of the reliable transport layer (that is, minimizes buffering) without relying on the help it might get from this layer. Thus we can migrate the flow control algorithm to UDP or use it to communicate with the clients who know nothing of this algorithm or the ones with an erroneous code, so not every request and response is delivered between them.

In order to achieve all these goals, the algorithm uses the simplest 'zig-zag' data receipt confirmation between the clients on the application level of the protocol. In case of the Gnutella network, PING messages with a TTL (time to live) of 1 are inserted into the outgoing data stream at the regular intervals (every 512 bytes). Such messages are broadcast only over 1-hop distance, so the peer should not broadcast them – it should just reply to them with PONGs (PING response) messages.

As soon as the sender receives such a response, the sender can be sure that all the data preceding the PING message has reached its intended destination, so it can send another 512 bytes with a PING. Thus we can be sure that at no moment the buffers of all the networking layers between servents contain more than 512 bytes+PING size + PONG size. Since the PING size in the Gnutella protocol is 23 bytes and the PONG size is 37 bytes, we are wasting only about 10.5% (See EQ. 2) of the whole bandwidth on these transmissions, which is not a huge amount, considering that the zig-zag schema occupies only about ½ of the available bandwidth in any case.

(2) (PINGsize+PONGsize)/(512+PINGsize+PONGsize)=60/572=10.5 % (EQ. 2)

This 'under-utilization' of the bandwidth is actually not an undesirable feature of the inventive method and algorithm -- would it happen to occupy the whole bandwidth, other applications (Web browsing, file upload/download) on the same computer would suffer. The flow control algorithm recognizes the fact that the distributed network routing servent is designed to work in the background with as little interference with the more important computer tasks as possible, so it tries to be unobtrusive.

The inventive method and algorithm also recognizes that the peer servent might know nothing of the flow control and/or be buggy (that is contain errors or bugs), so it might lose the PING altogether, might forward it to peers, sending back multiple PONG responses and so on. So the unique global ID field of the PING message is used to 'mark' the outgoing PINGS with their position in the connection data stream. Thus the duplicate and out-of-order PONGs can be rejected. Furthermore, if the reply does not arrive in 1 second (or some other predetermined period of time), the new PING (with a new sequence number) is sent in case the previous request or response was lost.

The inventive method and algorithm also limits the connection latency. For example, a client with a 33-40 kbits/sec modem and five connections will never have more than 3 Kbytes of data in transmission (572 times 5 plus headers), so even for this slow client the request delay won't be much more than about 900 ms (See EQ. 3) which is several times less than the typical latency value for the GNet without the flow control.

(3) 3Kbytes * 10 bits/byte / 33,000 kbits/sec = 900 ms (EQ. 3)

At the same time the inventive method and algorithm effectively limits the maximum possible data transfer rate on the connection. For example, if the Internet network round-trip time between servents is 150 ms, the connection won't be able to send more than 3400 bytes of data per second (140 kbits/sec per physical link if the servent has 5 connections) regardless of the physical link capacity. This feature can be viewed as a desirable one, but if it is not, this can be remedied by other means, such as for example by opening the additional connections, or by increasing the 512-byte data threshold between PINGs.

Note that if the underlying transport is TCP, the developer may be well advised to switch off the Nagle algorithm and send all the data in one operation (as one packet). Even though the algorithm will function in any case, this might allow the servent to get rid of the 200-ms additional latency added by the Nagle algorithm to the TCP stream. Since this might lead to an excessive packet traffic in case of the short roundtrip times and low sending rates, the outgoing flow control block implements the 'G-Nagle' algorithm. This is essentially a 200-ms timer, which is started when the packet is sent out. It prevents the next packet sending operation if all three of the following conditions are true: (i) the packet is not fully filled (less than 512 bytes), (ii) the previous packet RTT echo has already returned, and (iii) the 200-ms timeout has not expired yet. This algorithm is called 'G-Nagle' since it is basically a Nagle TCP algorithm ported into the Gnutella context. It serves the same goal of limiting the traffic with low payload-to-headers ratio if it is possible to do so without causing the perceptible latency increase.

Naturally, the flow control block routinely has to deal with the situations when other connections want to transmit some data, which cannot be sent because the PONG echo has not arrived yet and the 512-byte packet is already filled. In this case the behavior of the algorithm is determined by whether this data is the request (a broadcast) or the response (a routed-back message). The responses, which do not fit into the bandwidth, are buffered for the future sending – the goal is to keep the link latency to a minimum. It is the job of the Q-algorithm (described below) to make sure that these responses will be eventually sent out. The requests are just dropped according to their current hop count – the less is the hop count, the more important the request is considered to be.

This prioritization of the requests has two goals – first, it tries to make sure that the 'new', low-hop requests, which still exist only in a few copies on the network, are not dropped. If the zero-hop request is dropped, we lose a significant percent of this request copies and statistically are quite likely to drop all of them, thus making the whole search unsuccessful. And second, by dropping the high-hop requests, it tries to effectively introduce something like 'local TTL' into the network. We are effectively dropping the request with the hop count higher than some limit, but, unlike TTL, this limit is not preset or hard-coded. It is determined at run-time by the congestion state of the neighboring network segment. When we are doing this; we want to keep the average length of the search route to a minimum in order to minimize the network load and latency arising from the packet duplication on the way back. For example, it places a lighter load on the network to reach 4 servents with a 1-hop request over 4 connections, than to reach the same number of hosts with a 4-hop request over one connection. (The broadcast load is the same, but the first case results in 4 response messages with, say, 1-second latency, and the second case results in 1+2+3+4=10 messages transmitted over the network with a total latency of 4 seconds.)

Actually the precise algorithm defining which requests should be sent out and which should be dropped may be made a bit more complicated due to the competition between the requests coming from different connections. This scenario is further developed in the description of the Fairness block elsewhere in this description. The outgoing flow control block main goal is the latency minimization of the connection – not the fair connection bandwidth sharing between different logical substreams. However, it is worth mentioning that the TTL value of the Gnutella protocol loses its flow-control purpose in the presence of the flow control algorithm. It is desirably used mainly to avoid an infinite packet looping in case the servent code contains errors – pretty much in the same fashion as the IP packet TTL field is used by the Internet protocol (IP) networking layer code.

The responses (back-traffic) are also prioritized, but according to the total number of responses routed for their globally unique request ID. The idea is to pass the small number of responses at the highest priority without any delay, but if the request was formulated in a way, which resulted in a large number of responses, it is only fair to make the requesting servent wait. Since this request places an extra load on the network above the average level, it makes sense to send back its responses at the lowest priority only after the 'normal request' responses are sent.

The Q-Algorithm Block

The Q-algorithm block is logically situated on the receiving end of the connection. Its job is to analyze the incoming requests and to determine which of them should be broadcast over other connections (neighbors), and which should be dropped and not even passed to the outgoing flow control algorithm block described above.

The Q-algorithm block tries to limit the flow of broadcast requests so that the responses would not overflow the outgoing bandwidth of the connection, which has supplied these requests and will have to accept responses. Every response to route is typically a result of some significant effort of the network to broadcast the request, to route back the responses, and it would be wrong to drop the responses in the outgoing flow control block – it is more prudent just not to broadcast the requests. Besides, every response is a unique answer of some servent to some request. If the request is dropped, the redundant distributed network still might deliver the request to that servent over some other route with sufficient bandwidth. But if the response is dropped, the answer is irretrievably lost – the servent won't respond again, since it has already seen the request with this Global Unique ID (GUID).

So it is a job of the Q-algorithm block and the method provided therein to control the flow of the incoming requests so that the responses would not overflow the outgoing bandwidth of the same connection. It is especially important when the typical request generates several responses (highly available file is found) and/or the physical link is asymmetric – the connection cannot send as much data out as it can receive, which is typical for the ADSL and other high-bandwidth consumer-oriented ISP solutions. The excessive requests (the ones, which, if sent, would overflow the outgoing connections) are dropped according to essentially the same hop-priority algorithm as the one used by the outgoing flow control block.

Here it is important to note that in no case should Q-algorithm try to control the flow of the incoming requests **x** by slowing down the network reading. In fact, such an approach would likely be fatal for the flow control algorithm as a whole. First, it would ruin the algorithm backward compatibility in case of the Gnutella network – the flow control-unaware servents would keep sending data to the connection, which would be read with a slower rate. This would result in an explosive latency growth on the connection and possibly the connection would be dropped by the sender altogether.

Second, even in case of the flow control-compliant servents, who would use their outgoing flow control blocks to limit the connection latency, it should be remembered that the connection is shared between several streams of a very different nature. For example, the same connection is used to transfer both requests (which can be just dropped if the bandwidth is tight) and responses (which have to be delivered in any case to avoid the useless loading of the network). When the connection reading rate is throttled down, not only does the rate of the broadcasts from that connection decrease, but the rate of the response routing in the same direction also decreases. This response rate is something, which is supposed to be determined by the Q-algorithm on another servent and represents this other servent connection's back-stream. So there's no good reason why we should punish this back-stream just because our side of this connection has to throttle down the back-stream in the opposite direction.

Because of this, the output forward-stream flow **x** of the Q-algorithm should be formed as the subset of the incoming forward-stream **f** (requests, arriving to the connection). These requests should be read as fast as possible and only then those of them which exceed the required **x** value should be dropped. Unfortunately it may be very difficult to directly control the back-stream of responses by controlling the forward-stream of the requests. The direct application of the control theory fails for a variety of reasons, the main one being that the response for any particular request is not known in advance and its size and delay cannot be known with any degree of accuracy. The same request can result in zero responses and one hundred responses. The answers can arrive at once or be distributed over a 100-second interval – the rate cannot be predicted with any certainty. Besides, the volume of the back-traffic (responses) does not depend linearly on the amount of the forward-traffic (requests). After a certain broadcast rate is achieved, the outgoing flow control blocks on the same and/or other servents might take over and the further increases in the forward traffic won't affect the back-traffic at all. All this makes the 'normal' control theory inapplicable in that case.

This is why, in practical terms, the best one can hope for is to achieve the statistical control over the response traffic – say, '**x**' bytes per second of requests on the average generate '**b**' bytes per second of back-traffic responses. The ideal case would be that the back-traffic can be averaged and as the averaging interval grows the relative error (variation of the back-traffic distribution) converges to zero. Unfortunately, for all practical purposes this zero (or close to zero) relative variation of the back-traffic cannot be achieved since it turns out that statistically the distributions of responses on the Gnutella network exhibit the clearly pronounced fractal properties.

The fractal character of many network- and traffic-related processes is a well-known fact, demonstrated by many researchers See, for example, "*On the Self-Similar Nature of Ethernet Traffic (Extended Version)*" by W. Leland, M. Taqqu, W. Willinger and D. Wilson in IEEE/ACM TRANSACTIONS ON NETWORKING, Vol. 2. No. 1. February 1994; which is hereby incorporated by reference. It has been shown that the fractal, or self-similar distributions can be the result of the superposition of many independently functioning random processes with a 'heavy-tailed' distributions – that is, the distributions which asymptotically follow a power law. That is,

(4) P[X > x] ~ x ^ (-alpha) as x -> infinity, where 0 < alpha < 2. (EQ. 4)

(See, for example, "On the Effect of Traffic Self-similarity on Network Performance" by K. Park, G. Kim and M. Crovella in *Proceedings of the 1997 SPIE International Conference on Performance and Control of Network Systems;* which is hereby incorporated by reference).

Since the responses of Gnutella servents to the request are independent and the size of the response can range from zero (no matches are found) to a very large value (every servent responds), it is not a surprise that the back-traffic exhibits a fractal behavior.

From the mathematical standpoint, the self-similar random processes are characterized by the Hearst parameter H > ½ or the autocorrelation function of the form,

(5) r(k) ~ k ^ (-beta) as k -> infinity, where 0 < beta < 1. H = 1 – beta/2. (EQ. 5)

From the practical standpoint, the fractal traffic is characterized by its inherently bursty nature, which means that the bursts are clearly visible on very different time-scale plots and are not easily averaged as the averaging interval is increased (see the references above). It means that the back-traffic cannot be easily controlled so that its average would be close to the link (connection) bandwidth with a small variance. There's always a non-negligible chance that the traffic burst overloads the connection regardless of the averaging interval when a request with many responses is broadcast over the network. The results of the queuing theory suggesting that the mean latency starts to grow to infinity only as the link utilization (percent of the available bandwidth used to route back responses) approaches 1, do not apply for a fractal traffic. The latency growth starts with the loads much lower than the ones predicted by the queuing theory.

The results of the fractal traffic research (See for example, *"Experimental Queuing Analysis with Long-Range Dependent Packet Traffic"* by A. Erramilli, O. Narayan and W. Willinger; which is hereby incorporated by reference) suggest that an only way to keep the link latency acceptable is to keep the link utilization at about ½ level. That is, on the average, we should use only half of the link bandwidth to route the responses back.

It is to control this 'average back-traffic' that the Q-algorithm was introduced. Given the link bandwidth available for the back-traffic (responses) **B**, it tries to bring the average back-traffic rate **b** to the value, which would occupy the predefined portion of the available bandwidth **rho**:

(6) $$\langle b \rangle \text{ -> } \langle rho*B \rangle \qquad \text{(EQ. 6)}$$

For the sufficiently small values of **rho** (say, **rho** <= ½) that assures that the majority of the responses will be able to be routed back to the request originator without any delay whatsoever. Only the traffic bursts exceeding the target back-traffic value of **rho*B** by a factor of 1/**rho** (which is >= **2**), will cause the connection overload and the response latency increase, so this increase will affect only the small percentage of the responses. And even then, because of the back-traffic prioritization performed by the outgoing flow control block, the latency will be increased mainly for the requests with an unusually high number of responses.

Since the servent cannot control the back-traffic directly - only by controlling the amount of the forward-traffic (requests) **x** that it broadcasts - it is convenient to write our goal (EQ. 6) as:

(7) $$\langle x*Rav \rangle \text{ -> } \langle rho*B \rangle, \quad \text{(EQ. 7)}$$

where **Rav** is the estimated back-to-forward ratio; on the average, every byte of the requests passed through the Q-algorithm to be broadcast eventually results in Rav bytes of the back-traffic on that connection.

Furthermore, since the amount of the forward-traffic being broadcast is naturally limited by the rate of the requests arriving to the connection **f**, **x <= f**. So it can be that even when every arriving request is broadcast, still **x*Rav = f*Rav < rho*B**. This is why the new variable **Q** is introduced:

(8) $$Q = x*Rav + u, \; Q \; <= \; \langle B \rangle = Bav, \;\; \text{where} \;\; u = \max(0, Q - f*Rav) \; \text{(EQ. 8)}$$

Here **u** is the estimated underload factor. When **u > 0**, even if the Q-algorithm passes all the incoming forward traffic to be broadcast, it is expected that the desired part of the back-traffic bandwidth (rho*B) won't be filled. **u** is introduced into the equation to limit the infinite growth of the variable **x** when even the full forward traffic **f** won't fill the desired back-bandwidth. It allows to achieve the convergence of **x*Rav** to its desired value **rho*B** when **<f*Rav>** <= **<rho*B>** and limits its numerical growth with **f*Rav** otherwise. The algorithm tries to make **Q** converge to **rho*B + u** if it is possible:

$$(9)\qquad \langle Q\rangle \to \langle rho*B + u\rangle,\ Q <= \langle B\rangle \qquad \text{(EQ. 9)}$$

In order to achieve that, the Q-algorithm uses the stochastic differential equation:

$$(10)\qquad dQ/dt = -\,(beta/tauAv)*(Q - rho*B - u),\ \text{where} \qquad \text{(EQ. 10)}$$

$$(11)\qquad Q = x*Rav + u,\ Q <= Bav \qquad \text{(EQ. 11)}$$

$$(12)\qquad u = \max(0, Q - f*Rav) \qquad \text{(EQ. 12)}$$

to control the back-traffic. This equation causes the value of the variable **Q** to exponentially converge to the mean value of **rho*B + u** with a characteristic time of ~**tauAv** when the feedback coefficient **beta = 1**.

Note that the equations (EQ. 10 - EQ. 12) contain the random variables which cannot be controlled by the algorithm, like **B**, **Rav** and **f**. The Q-algorithm was specifically designed to avoid having the random and widely varying variables (like u) in the denominator of any equation. Otherwise, the stochastic differential equations of the Q-algorithm would exhibit some undesirable properties, preventing the algorithm output **Q** from converging to its target value.

The equations (EQ. 10 - EQ. 12) use the following conventions:

B – the link bandwidth available for the back-traffic.

rho – the part of the bandwidth occupied by the average back-traffic (1/2).

beta = 1 – the negative feedback coefficient.

tauAv – the algorithm convergence time. The relatively large interval chosen from the practical standpoint (100 seconds or so - more on that later).

Q – the Q-factor, which is the measure of the projected back-traffic. It is essentially the prediction of the back-traffic. The algorithm is called the 'Q-algorithm' because it controls the Q-factor for the connection. **Q** is limited with **<B>** to avoid the infinite growth of **Q** when **<f*Rav>** < **<rho * B>** and to avoid the back-stream bandwidth overflow (to maintain **x*Rav <= B**) in case of the forward-traffic bursts.

x – the rate of the incoming forward-traffic (requests) passed by the Q-algorithm to be broadcast on other connections.

f – the actual incoming rate of the forward traffic.

Rav – the estimated back-to-forward ratio; on the average, every byte of the requests passed through the Q-algorithm to be broadcast eventually results in **Rav** bytes of the back-traffic on that connection. This estimate is an exponentially averaged (with the same characteristic time **tauAv**) ratio of actual requests and responses observed on that connection (see (EQ. 14) below).

Bav – the exponentially averaged value of the back-traffic link bandwidth **B**. (**Bav = <B>**).

u – the estimated underload factor. When **u > 0**, even if the Q-algorithm passes all the incoming forward traffic to be broadcast, it is expected that the desired part of the back-traffic bandwidth (**rho*B**) won't be filled. It is introduced into the equation to limit the infinite growth of the variable **x** and ensure that **x <= f** in that case.

At any given moment the amount of incoming request traffic to pass to other connections is determined by the equation:

(13) x = (Q – u) / Rav = min(f*Rav, Q) / Rav (derived from (EQ. 10 - EQ. 12)). (EQ. 13)

Note that **x** from (EQ. 13) always obeys the rule **x** <= **f** – only the traffic actually arriving to the connection can be broadcast.

The predicted average back-to-forward ratio **Rav** is defined as an exponentially averaged instant value of the back-to-forward ratio **R** observed on the connection:

(14) dRav/dt = -(beta/tauAv)*(Rav – R) (EQ. 14)

One goal of the Q-algorithm block as defined by the equations (EQ. 10 - EQ. 14) is not to provide the quick reaction time to the changes in the back-traffic. In fact, in many cases it would be counterproductive, since it would cause the quick oscillations of the passed-through forward traffic **x**, making the system less stable and predictable. On the contrary, the Q-algorithm tries to gradually converge the actual back-traffic to its desired value **rho*B**, doing this slowly enough so that the Q-algorithm would be effectively decoupled from other algorithm blocks. This is done to increase the algorithm stability – for example, so that the non-linear factors mentioned above would not have a significant effect and would not cause the self-sustained traffic oscillations.

This determines the choice of the Q-algorithm averaging time **tauAv**. First of all, this time should not be less than the actual time **tauRtt** passing between the request being broadcast and the responses being received, since it would not make any sense from the control theory standpoint. The control theory suggests that when the controlled plant has the internal delay **tauRtt**, it is useless to try to control it with the algorithm with characteristic time less than that. Any controlling action will not have effect for at least **tauRtt** and trying to achieve the control effect sooner will only lead to the plant instability.

However, the network reaction time **tauRtt** can be small enough (seconds) and we might want to make **tauAv** much bigger than that for the reasons outlined above – from the purely fractal traffic standpoint, it is ideal to have an infinite averaging time. On the other hand, it is impractical to have an infinite or even very large averaging time, since it limits the speed of the algorithm reaction to the permanent changes in the networking environment. For example, when one of the connections is closed and reopened to another host with a different bandwidth, the random process **R**, which defines the instant back-to-forward ratio is irretrievably replaced by the new one with another theoretical mean value **<R>**. At this point we need the equations (EQ. 10 -

EQ. 14) to converge to the new state, which is possible only if the algorithm averaging time is much less than the average connection lifetime.

For the Gnutella network the connection lifetime is measured in hundreds of seconds, so the averaging time is chosen to be:

(15) tauAv = max(tauRtt, tauMax), where tauMax = 100 sec. (EQ. 15)

At the same time it is important to remember that in case the back-traffic overloads the connection and it starts to buffer the responses, the Q-algorithm can cut the flow of the back-traffic and to decrease the overload only after the **tauAv** time interval. In the meantime, if the overload is caused not by a short burst, but rather by a permanent network state change, the back-traffic will continue to be buffered, increasing the effective response delay. In order to mitigate this, the averaging interval **tauAv** is made as small as reasonably possible (that is, **tauRtt**) when the mean back-traffic **<b>** exceeds the mean connection bandwidth **<B>**.

The mean values for the back-traffic and for the connection bandwidth are approximated by the exponential averages **bAv** and **Bav**, which are calculated with the equations similar to (EQ. 10) and (EQ. 14):

(16) dbAv/dt = -(beta/tauAv)*(bAv – b) (EQ. 16)

(17) dBav/dt = -(beta/tauAv)*(Bav – B) (EQ. 17)

Thus the short back-traffic bursts do not cause the change in the **tauAv**, but the long bursts, which might indicate the permanent change in the network configuration cause the **tauAv** to be equal to **tauRtt**. This causes the Q-algorithm to quickly converge to the intended back-traffic value, after which (when **bAv** becomes smaller than **Bav**) the regular large value of **tauAv** takes over and the short traffic bursts are effectively ignored - as they should be from the fractal traffic model standpoint.

So the final expression for the Q-algorithm averaging time **tauAv** is:

(18) tauAv = max(tauRtt, tauMax), if bAv <= Bav and (EQ. 18)

$$\text{tauAv} = \text{tauRtt} \qquad \text{if} \qquad \text{bAv} > \text{Bav}.$$

It is noted that even though the outgoing flow control block of the flow control algorithm is intended to be used only with the connections, which use the reliable transport at some level, the Q-algorithm has no such limitation. The Q-algorithm might be useful even when the connections do not have any reliable component whatsoever – for example, when the broadcast is implemented as the global radio (or multicast) broadcast to all receivers available. In this case the Q-algorithm would decrease the total network load by limiting the broadcast intensity to the level at which the majority of the responses can be processed by the network.

So far we have not discussed in detail the method of determining of the outgoing connection bandwidth used to route the back-traffic. As far as the Q-algorithm is concerned, the numerical value of **B** is just another input parameter provided to it by some other algorithm block. In case of the connections utilizing the reliable transport at some level in general and the Gnutella connections in particular, this value is determined by the fairness block of the flow control algorithm.

Fairness Block

The Fairness block is logically situated at the sending end of the connection and is a logical extension of the outgoing flow control block. Whereas the outgoing flow control block just makes sure that the outgoing connection is not overloaded and its latency is kept to a minimum, it is the job of the fairness block to make sure that: (i) the outgoing connection bandwidth available as a result of the outgoing flow control algorithm work is fairly distributed between the back-traffic (responses) intended for that connection and the forward-traffic (requests) from the other connections (the total output of their Q-algorithms); and (ii) the part of the outgoing bandwidth available for the forward-traffic broadcasts from other connections is fairly distributed between these connections.

The term fairly distributed as used here means that when some bandwidth is shared by the multiple logical streams, no stream or a group of streams should be able to monopolize the bandwidth, effectively starving or denying access by some other stream or a group of streams and making it impossible for them to send any meaningful amount of data.

Some mathematical notation, variables, and conventions are now introduced that will be used in subsequent description. Let **i** be the connection number on the servent. For example, if the servent has five connections, 1 <= **i** <= 5. This index is used to mark all the variables related to that connection:

Gi – The total outgoing connection bandwidth.

Bi – The part of the **Gi** 'softly reserved' for the back-traffic. This means that when the back-traffic **bi** is less than or equal to **Bi**, it is unconditionally sent out without any delays.

bi – The outgoing back-traffic (responses) on that connection.

fi – The forward traffic sent from this connection to other connections to be broadcast. This is essentially equal to the output of the Q-algorithm **x** described elsewhere herein.

yi – The total desired forward-traffic from the other connections – this is what they would like to be broadcast if there's enough bandwidth available. So **yi = sum(fj | j != i)**, plus the requests that are generated by this servent (not received by it to be broadcast).

boi – The incoming response traffic intended for the other connections. **bi = sum(boj | j != i)** + the responses that are generated by this servent (not received by it to be routed) if the connection has enough bandwidth to route all the responses. Otherwise the **bi** value might fluctuate because of the response bufferization by the outgoing flow control block.

foi – The outgoing request traffic sent out by this connection. It shares the outgoing connection bandwidth with **bi**, so **foi = min(yi, Gi – bi)**. This means that if the connection can send all the other connections' Q-algorithms outputs **fj**, then **foi = yi**; otherwise **foi** is limited by the connection bandwidth available and the outgoing back-traffic **bi**: **foi = Gi – bi**.

di – The part of the total desired forward-traffic **yi** dropped by the algorithm when it cannot send all of it. **di = yi – foi = max(0, yi + bi – Gi)**.

Thus the total outgoing connection bandwidth **Gi** is effectively broken into two sub-bands: **Bi** and **Gi – Bi**, where 0 < Bi < Gi and 0 < Gi – Bi < Gi. **Bi** is the part of the total bandwidth 'softly reserved' for the back-stream **bi**, and **Gi – Bi** – is the part 'softly reserved' for the forward-stream **foi**. (Here all the traffic streams do not include the 1-hop messages used by the outgoing flow control block to limit the connection latency – they are regarded as an invisible 'protocol overhead' by this block).

The term 'softly reserved' as used here means that when, for whatever reason, the corresponding stream does not use its part of the total bandwidth, the other stream can use it, if its own sub-band is not enough for it to be fully sent out. But if the stream **bi** or **yi** cannot be fully sent out, it is guaranteed to receive at least the part of the total outgoing bandwidth **Gi** which is 'softly reserved' for this stream regardless of the opposing stream bandwidth requirements. For brevity's sake, from now on, we will actually mean 'softly reserved' when we will apply the word 'reserved' to the bandwidth.

Let's imagine that we have some method of finding an optimal **Bi** and see how would the traffic streams behave themselves in that case. In the long run, the Q-algorithm would make sure that the mean value of **bi** obeys the rule:

(19) <bi> <= <Bi>/2, (since rho = ½) (EQ. 19)

Here <bi> < <Bi>/2 if the incoming stream of requests on the connection is not powerful enough to fully occupy one half of the bandwidth **Bi** and **<bi> = <Bi>/2** otherwise.

This means that on the average at least one half of the total outgoing bandwidth **Gi** is available for the other-connections forward traffic **foi**. So on the average the mean value of the forward traffic from other connections **<foi>** will be non-zero:

(20) 0 < <foi> <= <Gi – bi> (EQ. 20)

Now let's imagine that for some reason the back-traffic **b** experiences a burst which fully occupies the bandwidth **Bi**, and at the same time the di > 0 (other connections wish to send more requests than what would fit into the bandwidth). The fairness requrement means that the forward-traffic streams, which have nothing to do with the response traffic burst, would not be greatly affected. In any case, this back-traffic burst should not bring the connection bandwidth dedicated to the forward-traffic to zero.

In fact we require that the back-traffic burst should not decrease the forward-traffic on the same connection by more than a factor of two. Then we can easily write the equation for the mean value of **Bi**:

$$(21) \qquad \langle Gi - Bi\rangle = \tfrac{1}{2}*\langle foi\rangle, \quad \text{or} \quad \langle Bi\rangle = \langle Gi - \tfrac{1}{2}*foi\rangle \qquad \text{(EQ. 21)}$$

This equation can be strict only when the averaging interval is an infinite one. In reality, we need the mean value of **Bi** to converge to its ideal value:

$$(22) \qquad \langle Bi\rangle \text{ -> } \langle Gi - \tfrac{1}{2}*foi\rangle = \langle Gi - \tfrac{1}{2}*\min(yi,\ Gi - bi)\rangle = \langle Gi - \tfrac{1}{2}*(yi - di)\rangle \qquad \text{(EQ. 22)}$$

If the total outgoing connection bandwidth is known, we can immediately arrive from this to the differential equation for the **Bi** in the same fashion as we have arrived to the equation (EQ. 10) from (EQ. 9) in the Q-algorithm:

$$(23) \qquad dBi/dt = -(beta/tauAv)*(Bi - Gi + \tfrac{1}{2}*foi)\ . \qquad \text{(EQ. 23)}$$

Unfortunately, in practice the value of **Bi** calculated with the help of that equation is essentially unusable – **Bi** is the slowly changing averaged value, and **Gi** is a quickly changing random input. So it is quite possible to have **Bi > Gi**, which leaves no bandwidth whatsoever for the forward-traffic at that moment.

In order to solve this problem, the new variable **ri** is introduced. **ri** is the share of the total bandwidth **Gi** reserved for the back-traffic:

(24) $Bi = ri * Gi$ (EQ. 24)

Then the expressions (22) and (23) can be written as:

(25) $<ri> \rightarrow <1 - \frac{1}{2}*foi/Gi>$ (EQ. 25)

and

(26) $dri/dt = -(beta/tauAv)*(ri - 1 + \frac{1}{2}*foi/Gi)$ (EQ. 26)

respectively. It is easy to see that **ri** calculated from the equation (26) cannot exceed 1. One can take the slowly changing value of **ri**, multiply it by the current fast-changing value of **Gi** and arrive to the value of **Bi** to feed into the Q-algorithm which is guaranteed to be less than **Gi**.

The variable **ri** also makes it simpler to implement the bandwidth sharing. To guarantee that as we do the send the back-traffic sending rate **bi** would not exceed the value of **Bi** defined as **ri * Gi**. Before we perform the actual sending operation, it might be very difficult to determine the precise value for **Gi**. Even if we would know the exact link characteristics between the servent and its peers and would know the hardware bandwidth between them, there are several factors that can affect this value. First, the same hardware link is shared by several connections. Second, the outgoing flow control block significantly decreases the maximum theoretical throughput when it tries to minimize the connection latency. And third, there might be many other independent processes sharing the same hardware link – Web browsing, FTP transfers, and the like.

In fact, the only thing one may generally know about the connection is that the last time **V** bytes of data were sent, it took the outgoing flow control 'echo PONG' **T** seconds to arrive back. Theoretically it might be possible to create the data transfer model which would be able to predict the time **T** given the transfer amount **V**. But in practice such models are very imprecise, noisy and generally unreliable. It is much easier to abandon the **Gi** prediction altogether and use **ri** to directly calculate the number of bytes in the total packet sent out when the 'echo PONG' arrives. If the total packet size is **V** (for example, **V** = 512 bytes when the outgoing flow control block has a large amount of data to send), we just have to allocate

(27) $Vb = ri*V$ (EQ. 27)

bytes for the back-traffic data (responses) in this packet. Then regardless of what would be the value of **T** after which the echo will arrive, the resulting bandwidth value will be **Gi = V/T**, and the **Bi = ri*V/T**, which gives us exactly the value **Bi = ri*Gi** that we need.

Note that the **Bi** estimate for the Q-algorithm as **Bi = Vb/T** is not precise but the error has been modeled and shown not to be prohibitively large. In the worst case, when the estimate error reaches its maximum, the mean back-traffic **<bi>** converges to about **<0.707*Bi>** instead of **<0.5*Bi>**, and even this relatively small error can be observed only when **<foi> -> 0**, which is not a very common situation for the real-life networks.

A second half of the fairness algorithm is now described, which assures the 'fair sharing' of the outgoing forward-stream **foi** between the request streams from the different connections. This algorithm block is very important in a practical sense, since it allows the coexistence of very-different-bandwidth connections on the same servent and is also largely responsible for the whole flow control algorithm ability to withstand the Denial-of-Service (DoS) attacks.

Unfortunately the full mathematical model of the servent behavior in case of the different-bandwidth links or a servent under the DoS attack is complex and is not described here in detail. In fact, an attempt to arrive to the analytical solution of the corresponding system of equations might be counterproductive. The reason for this is that this system contains about ten equations for every servent connection (the equations (EQ. 10 - EQ. 12), (EQ. 23), (EQ. 26) and some additional equations describing the interaction between the connections). The interpretation of the analytical solution even for five connections or so is prohibitively complex. This is why the servent behavior under these conditions was analytically modeled only in a steady-state approximation and numerically – for some simple cases (for example, the connections were divided into two groups, and all the connections in a group shared the similar characteristics).

Still, even in that case the full presentation of the results would take a lot of space, so this section of the document just presents the final results - the algorithm itself and some common-sense explanations of why it has been chosen.

Consider the servent A connected to the servent B and to some other servents C, D, E, and so on. Now consider the 'fairness block' of the connection, which links A to B and use index **i** to denote the variables related to that connection. At any given moment, if the summary flow of forward-traffic from other connections **yi** is less than or equal to **Gi – bi**, the bandwidth sharing is not an issue, since everything can be sent out anyway.

The problem of the fair forward bandwidth sharing arises only when **yi > Gi – bi**, where the value of **bi** is calculated by the forward/backward fair bandwidth sharing algorithm described above. Recall, **yi** was defined as **yi = sum(fj | j != i)**, plus the requests that are generated by the servent A (not received by it to be broadcast). If **yi > Gi – bi**, some requests have to be dropped from the streams **fj** in order to bring the total forward-traffic on the A-B connection to the value of **foi = Gi – bi**. The outgoing flow control algorithm block requires the requests to be prioritized according to their hop count, so the fair sharing algorithm should prioritize the requests from the different connections but having the same hop count. (One consequence of this approach is that the requests from the servent A itself will always have the highest priority if they are present, since these are the only requests with a hop count of zero.)

For a practical implementation, one may actually consider it to work in terms of the part **Vf** of the total packet volume **V** to be sent out. **Vf** is the part of the packet dedicated to the forward-traffic in a pretty much the same fashion as **Vb** was dedicated to backward-traffic in (EQ. 27). The reason for that approach would also be similar - the precise values of **Gi, bi**, etc are not known at the time of the packet sending. Still, here we will keep using the **Gi, bi, foi** variables for the illustrative purpose, even though in practice they are likely to be replaced by the variables **V, Vb, Vf** with little or no effect on the algorithm operation.

Some additional variables are now introduced, let **fjk** designate the part of the connection **j** incoming request stream which has the hop **k** and **foik** – to designate the part of the outgoing forward-traffic stream of the connection **i**, which carries the requests with a hop count value of **k**. The requests from the servent A itself are regarded as the requests with connection number **j=0**, so they can be regarded as the regular part of the stream. The only special feature of this sub-stream with **j=0** is that it carries only the requests with a zero hop count, which does not affect the treatment of such requests by the fairness algorithm – they don't receive any special priority. The fairness algorithm is supposed to be designed in a way which would send these requests first even without any knowledge about their importance. In the situation we are interested in, that is when the fair bandwidth-sharing algorithm has to be invoked, it follows that:

(28) sum(foik | 0 <= k <= maxTTL) … < sum(sum(fjk | j != i) | 0 <= k <= maxTTL) (EQ. 28)

Thus at least some of the requests in the streams **fjk** have to be dropped, and it is the job of the fairness algorithm to decide what is dropped and what is added to the outgoing streams **foik** to be sent out.

Before we describe the fairness algorithm in further detail, let's consider a few examples to illustrate how it desirably should not work. Naturally, the fairness algorithm should not throttle down any connection's stream to zero as it drops the requests. So, for example, it should not operate by taking all of the requests from connection 0, then 1, 2, … and dropping all the rest (connections m…N) when the bandwidth limit is reached. This approach would just arbitrarily choose the connections with high numbers and stop forwarding all traffic from them. (This hardly seems 'fair'.) Similarly, it should not keep taking the random requests from the streams **fj | j != i** until the bandwidth limit **Gi – bi** is reached. Statistically that approach would seem to give every connection's stream a non-zero bandwidth, but in practical terms, if some connection would generate a very high-rate request stream, its requests would occupy most of the bandwidth, leaving almost nothing for all the other connections. Specifically, if that high-rate stream would be a result of the DoS attack, the attacker would effectively reach its goal, preventing the servent A from broadcasting any requests than the ones generated by the attacker and bringing all the useful traffic through this host close to zero. The same objection is true for the method, which shares the outgoing stream **foi** between the request streams from other connections in proportion to their desired sending rate **fj**, regardless of the method used to prioritize and to drop the different-hop requests **fjk** within the **fj** stream itself.

This is why the fairness block implements the 'round-robin fairness algorithm'. This algorithm works as follows: first, for every connection with number **j != i**, it prioritizes the requests it wishes to send according to their hop count values with small hop counts having the highest priority. This operation forms N logical queues, where N is the number of the servent A connections. (N-1 for the connections with number **j != i** and one queue for the 'fake' connection with number zero, which represents the servent A requests). Then it starts taking data from these logical queues' heads in a round-robin fashion and transfers this data to the buffer to be sent out on connection **i** . This transfer is being performed in a 'hop-layer' fashion, meaning that the requests with the hop count **k+1** are not touched (the corresponding connections are just skipped) until all the requests with the hop count of **k** has been transferred. Thus regardless of how much data the connection **j** wishes to broadcast on the connection **i**, when the send buffer is full (the **Gi – bi** bandwidth limit is reached), only a volume of data comparable to the other connections' volumes will be actually sent.

Actually, the servent can perform a DoS attack and send a large amount of requests with a hop count of zero, effectively preventing its neighbors from broadcasting anything but these requests. But these neighbors will be able to send their own requests without any problems whatsoever, since the attacking requests will have a hop count of one and have the lower priority. The servents one more hop further from the attacking servent will be able to issue their own requests and to route the legitimate requests with a hop count of 1. The detailed examination of the traffic flows in such a case shows that such a DoS attack will weaken as the distance from the attacking host grows, and in no case will it significantly affect the user-perceived performance of the servents under attack.

It is desirable that this round-robin algorithm to transfer an equal number of bytes and not an equal number of requests from every logical queue. Otherwise, a DoS-attack is possible when the attacker issues very large (several kilobytes) requests, which effectively take over the whole outgoing forward-traffic bandwidth and prevent the useful requests from being sent.

One other way of implementing the round-robin fairness algorithm is to find the threshold values for the hop **m** and for the sending rate **ft** . The hop is chosen so that all the request substreams **fjk** with hop value **k < m** fit into the **Gi – bi**, and the request substreams with hop value **k <= m** do not fit. The sending rate **ft** is calculated for sum of the 'threshold hop' **m** request substreams **sum(fjm | j != i)**. **ft** should allow the request substreams with the rate **fjm <= ft** to be fully sent out, the streams with the rate **fjm > ft** to be sent out only at the rate **ft**, and the total resulting stream to be equal to **Gi – bi**:

(29) Gi – bi = sum(foik | k < m) + sum(fjm | fjm <= ft) + ft * n, (EQ. 29)
where n is the number of connections with fjm > ft.

This way might be preferred since it does not require the full sorting of the request queues according to the hop count, but might be difficult to implement in case of the large-size requests. However, the implementation of the discrete-traffic (finite request size) fairness algorithm might be tricky regardless of which algorithm variant is used and is outside of this document scope in any case. For example, one might want to make sure that the round-robin operation does not start from the same connection every time to avoid the misbalance between the connections, but still might give a priority to the "A's own" requests from the 'pseudo-connection' number zero.

Finally it is worth mentioning why the round-robin fairness algorithm works in a 'hop-layer' fashion rather than just taking the data from the connection queues regardless of their hop. The reason for this is that otherwise a connection with a low-rate input request stream would have even the high-hop requests broadcast, whereas the high-rate connection would have even the low-hop requests dropped. That would be unfair to the servents sending the requests through the network segments with the high-capacity links in the request routes. Such clients would have the 'shorter reach' - their 'visibility radius' would be lower, meaning that their requests would reach the lower number of servents than average. Since it was assumed to be wrong to punish the requestor for having the high-bandwidth link close to it in the network, the hop-layer round-robin procedure has been provided.

<u>Discrete Traffic Case</u>

Having now described embodiments of a flow control method and algorithm for a distributed broadcast route type networks in a continuous traffic approximation – it has ben assumed that the traffic streams do not have a 'minimum transfer unit'. Actually the requests and responses may not be infinitely small, and in fact can be very large. Note that Gnutella protocol specifies a limit of as high as 64 Kbytes for the request and the response size.

This fact has at least one noteworthy consequence for the practical implementation of the inventive method and algorithm. Some of these consequences were already mentioned, but the consequence of very large packet treatment deserves some additional amplification. The Gnutella protocol multiplexes several logical streams over a single TCP connection by sending the sequence of requests and responses belonging to the different substreams. This means that when a very large request or response is being pushed through the wire or other communication channel, nothing else can be transmitted over the same connection until it goes through. At the same time, the latency of all other connections on the same physical link goes significantly up even in the presence of the outgoing flow control block. This gives one an idea of the DoS attack utilizing the very large packets to bring the latency of the neighboring hosts to the unacceptable levels. Unfortunately, nothing can be done to fully deflect this attack until the Gnutella protocol is changed to use some other multiplexing method, which would allow the large requests to be sent out in small chunks.

In the meantime, the flow control algorithm described here should not broadcast or route the requests and responses bigger than some reasonable size (3 Kbytes or so). One might also try to close the connections to the servents, which try to send the messages of twice that size (5-6 Kbytes), since the latency on such connections would be too high for normal operation anyway, but such a behavior would open the door for the 'old-servent DoS attack'. In this attack, the malicious host might try to broadcast the large requests through the 'old' servents, which are not aware of the flow control algorithm. Thus the attacker can hope to eventually reach many 'new', flow-controlled servents and remotely terminate all their connections to the 'old' ones. So in the absence of the proper low-latency stream multiplexing in the protocol it is preferable to just drop the big messages without closing the connection. Providing a smooth transition to the flow-controlled protocol without breaking the Gnutella network in the process may also be considered.

The foregoing descriptions of specific embodiments of the present invention have been presented for purposes of illustration and description. They are not intended to be exhaustive or to limit the invention to the precise forms disclosed, and obviously many modifications and variations are possible in light of the above teaching. The embodiments were chosen and described in order to best explain the principles of the invention and its practical application, to thereby enable others skilled in the art to best utilize the invention and various embodiments with various modifications as are suited to the particular use contemplated. It is intended that the scope of the invention be defined by the claims appended hereto and their equivalents. All publications and patent applications cited in this specification are herein incorporated by reference as if each individual publication or patent application were specifically and individually indicated to be incorporated by reference.